CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Resource Optimal Neural Networks for Safety-critical Real-time Systems

Master's thesis in Computer science and engineering

Joakim Åkerström

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Resource Optimal Neural Networks for Safety-critical Real-time Systems

Joakim Åkerström

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Resource Optimal Neural Networks for Safety-critical Real-time Systems

Joakim Åkerström

Supervisor: Selpi Selpi, Department of Mechanics and Maritime Sciences
Advisor: Vedad Cajic & Srikar Muppirisetty, Volvo Cars Corporation
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Resource Optimal Neural Networks for Safety-critical Real-time Systems

Joakim Åkerström
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Deep neural networks consume an excessive amount of hardware resources, making them difficult to deploy to real-time systems. Previous work in the field of network compression lack the explicit hardware feedback necessary to control the resource constraints imposed by such systems. Furthermore, when the system under discussion is safety-critical, additional constraints must be enforced to make sure that acceptable safety levels are achieved. In this work, we take a reinforcement learning approach with which we evaluate three different compression actions: filter pruning, channel pruning and Tucker decomposition. We found that channel pruning was the most consistent one as it satisfied the constraints specification on five of six test scenarios while providing compression and acceleration rates of 10-30% across most resource metrics. By further optimizing the networks with TensorRT, we managed to improve the resource efficiency of the reference networks by up to 6×.

Keywords: Data science, machine learning, deep learning, neural networks, network compression, network acceleration, safety-critical systems, real-time systems.

Acknowledgements

I want to thank Volvo Cars for giving me the opportunity to work on a very exciting topic over the past couple of months. Special thanks goes to my advisors, Vedad Cajic and Srikar Muppirisetty, whose expertise has been invaluable for the completion of this project. I also want to thank my supervisor Dr. Selpi for excellent academic supervision throughout this thesis work.

Joakim Åkerström, Gothenburg, June 2020

Contents

1
Introduction

Deep neural networks (DNNs) provide state of the art solutions to many computer vision tasks, e.g. object recognition, where they have shown human-level performance on various benchmarks [10]. That said, a well-known limitation of these models is their large memory consumption. While this can be a problem in itself for small devices, it often causes additional problems such as high-latency inferences and energy inefficiencies, due to the large amount of memory transfers needed for data propagation. These issues make it difficult to deploy DNNs to real-time systems which need to guarantee certain response times for computational operations, typically under a very tight resource budget.

Established techniques for network optimization, including network compression [14, 18, 22] and knowledge distillation [19, 1], have demonstrated the possibility to alleviate aforementioned issues by reducing the size of pre-trained networks. Recent studies in this area have shown that significant reductions and speedups can be achieved with little or no loss of predictive performance. However, these techniques have mostly been developed and evaluated in the context of mobile systems, where the constraints are slightly different from those of real-time systems. In particular, since real-time systems need to guarantee acceptable response times, they need the ability to reduce the size of a network dynamically whenever it fails to satisfy those guarantees. This requirement adds complexity constraints to the optimization process which, to our knowledge, have not been considered in previous works. Furthermore, when the system under discussion is safety-critical, additional constraints must be incorporated into the optimization problem to ensure that safety-level thresholds are met.

Examples of safety-critical real-time systems include those of advanced driver-assistance systems (ADAS). Such systems will typically employ multiple DNNs, each trained for a specific task such as pedestrian detection [35] or scene segmentation [27], to interpret the surroundings of the vehicle. Given the limited computational resources available in the vehicle, running multiple DNNs simultaneously is difficult. On the other hand, not all of those DNNs are safety-critical during the entire driving cycle. This allows for dynamic trade-offs between different model requirements (e.g. accuracy, latency, throughput, memory footprint, energy consumption and computational operations) to achieve the system requirements.

1.1 Problem

The great modeling capacity of DNNs is mainly attributed to their large number of learnable parameters, which effectively enables them to extract very complex patterns in high-dimensional feature spaces [12]. From a computational perspective, however, the large number of parameters can be troublesome. Not only do they have to be stored somewhere, ultimately leading to a large memory footprint; they must also be loaded onto the computational device, where they are transferred between *streaming multiprocessor* (SM) caches to participate in arithmetic operations. All these operations take a considerable amount time and energy, which inhibits the deployment of DNNs to real-time systems operating in resource constrained environments.

Meanwhile, it has been theorized that the large number of parameters is only needed for the training phase; once the data patterns have been recognized, an overwhelming proportion of the total parameters can be considered superfluous. This theory is sometimes referred to as the *overparameterization dilemma* [28, 8], which underlies the research field of neural network optimization.

A concrete illustration of the overparameterization dilemma, as well as a proposed solution, is given by *deep compression* [14]. In this paper, a three-stage pipeline of weight pruning, trained quantization and Huffman coding is used to reduce the number of superfluous connections in pre-trained DNNs. Results showed that this compression pipeline can reduce the size of DNNs by up to 49× with no loss of classification accuracy. The large size reductions allowed the models to fit onto mobile-sized *static random-access memory* (SRAM) caches, which can be accessed much more cheaply than *dynamic random-access memory* (DRAM). The authors noted that this may lead to significant savings in inference latency and energy consumption, but extensive evaluations of these metrics were not made.

While deep compression gives an illustration of the problem and a rough indication of the possible gains, it also illustrates a few limitations that are common to most of the previous work in this field. First, it uses extensive fine-tuning after the pruning and quantization stages which prohibits fast, dynamic compression during the execution of the host system. For real-time systems, such a capability is essential in order to alter the model according to the instantaneous availability of computational resources. Even if one could afford this fine-tuning from a computational perspective, the original dataset used to train the model may not be accessible due to various reasons. Secondly, the authors of deep compression use model size (counted as the number of parameters) as the sole evaluation metric. While this metric is most likely correlated with other metrics such as latency, throughput and energy consumption, all of which require careful monitoring in real-time embedded systems, the extents of those correlations are unknown. Lastly, the authors provide no clue as to how their solution can be integrated into continuous integration (CI) and continuous development (CD) workflows, which is crucial for an efficient deployment.

1.2 Objective

The high-level goal of this project is to explore methods to incorporate network optimization into safety-critical real-time systems. As highlighted in the problem statement, such systems introduce constraints that have not been considered in previous works. Real-time systems need to guarantee certain response times for computational operations. As such, they need the ability to optimize networks at runtime according to the availability of hardware resources and other system requirements. In order to justify such a model switch, the optimization process needs to be fast, which prohibits the usage of fine-tuning and other expensive reconstruction methods. Furthermore, since real-time systems typically operate in resource constrained environments, the array of performance metrics for which to optimize the network needs to be larger than in previous works [14]. Specifically, *direct* metrics such as latency, throughput and energy consumption should be accessible to the optimization process to evaluate a candidate solution.

Safety-critical systems, on the other hand, need the ability to guarantee a certain degree of safety in their operations. Network optimization can assist the system with enforcing this guarantee, by allowing it to redistribute its computational resources to the more safety-critical operations. Obviously, this assumes that the optimization can be done with predictable changes in modeling performance (e.g. accuracy, precision and recall). In particular, the optimization solution needs to support both soft and hard constraints. Finally, we want to explore the options for integrating network optimization techniques into continuous software development processes, in a safe and efficient way. To concretize these goals, the top-level research question to be answered is formulated as follows:

> *How can neural network optimization be incorporated into safety-critical real-time systems?*

Since this question is rather large and hence difficult to answer in a definitive way, we split it up into the following subquestions:

- Q1 *What is an appropriate network optimization objective for safety-critical real-time systems?*
- Q2 *How do different optimization algorithms perform according to the objective determined in (Q1)?*
- Q3 *How can the algorithms compared in (Q2) be integrated into a continuous software development process?*

It should be noted that (Q3) is still too large to be answered thoroughly within the time frame of this project. As such, we have placed an extra emphasis on answering (Q1) and (Q2), while addressing the issue posed by (Q3) in the form of a discussion.

1.3 Scope

The full solution space to our objective is too large to be exhaustively explored within the time constraints of this project. Hence, to increase the feasibility of the project, a few limitations have been imposed. First, the work is targeted to systems which consists of multiple networks, not all of which are safety-critical at all times. Secondly, we have focused on the optimization of *convolutional* neural networks (described in Section 2.2.1), as they are ubiquitous in object-detection systems which are often safety-critical. Lastly, in the vast field of neural network optimization, we have focused on the branch of network compression. We feel that this is the most viable branch for the problem at hand. Furthermore, while other branches such as knowledge distillation could be viable in some circumstances, it is not so easy to compare such solutions with those of network compression.

1.4 Outline

The remainder of this thesis is structured as follows: Chapter 2 and 3 describe the theory underlying this work, Chapter 4 describes the methods used to solve the underlying research problem, Chapter 5 reports the obtained results, Chapter 6 discusses those results and Chapter 7 presents the conclusions derived from this work.

2
Theory

This chapter introduces the background material necessary to follow the remainder of this text. A short introduction to computer vision in general, and image classification in particular, is given in Section 2.1. Neural networks, which are commonly used to solve image classification and other computer vision tasks are described in Section 2.2. Lastly, Section 2.3 gives an overview of reinforcement learning, which is central to the methods used to solve the problem of this work.

2.1 Computer Vision

Computer vision refers to the idea of building machines that can recognize high-level patterns in visual data, such as images and videos. Video tracking, object recognition and pose estimation are examples of tasks that this discipline is concerned with [12]. In this work, we focus on image classification, where the task is to find a mapping of the form:

$$X \mapsto Y \tag{2.1}$$

where X is a space of images and Y is a set of discrete classes. In practice, an image is typically represented as a three-dimensional tensor of fixed size. Hence, we can rewrite (2.1) as:

$$\mathbb{R}^{w \times h \times c} \mapsto Y \tag{2.2}$$

where w and h represent the width and height of the images, respectively, and c represents the number of color channels (e.g. three for red, green and blue). Once such a mapping has been found, it can be used to *infer* the classes of new images. One famous image classification task is the annually recurring *ImageNet large scale visual recognition challenge* (ILSVRC) [29]. The ImageNet dataset consists of around 1.3 million images evenly distributed across 1,000 classes which describe the content of the images. While the ImageNet dataset was primarily compiled for ILSVRC, it is often used to benchmark novel image classification methods proposed by public research. Methods to solve image classification problems, on ImageNet and other datasets, are often based on convolutional neural networks, described in Section 2.2.1.

2.2 Neural Networks

A neural network can be viewed as an optimizable approximator to a nonlinear function $y = f^*(x)$. This makes them popular for solving classification tasks where the goal is to find a mapping between x and y [12]. There are different types of neural networks, but the quintessential type is the *Feedforward neural network* (FFNN). This type of network contains one or many *layers*, each taking an input $x \in \mathbb{R}^n$ and produces an output $y \in \mathbb{R}^m$ by applying a nonlinear activation function on a weighted combination of x:

$$y = f(Wx) \tag{2.3}$$

where $W \in \mathbb{R}^{m \times n}$ is a weight matrix and f is a nonlinear activation function which is applied to an input vector component-wise. From here on, the weighted combination Wx will be referred to as the *Generalized matrix multiplication* (GEMM) of W and x. Common choices for the nonlinear function f include:

$$\text{sigmoid}(z) = \frac{1}{1 + e^{-z}} \tag{2.4}$$

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.5}$$

$$\text{relu}(z) = \max(0, z) \tag{2.6}$$

Sometimes, the input and output of a network are also regarded as layers. In those cases, the computational layers are usually referred to as *hidden* layers. In the remainder of this text, we use the terms layer and hidden layer interchangeably whenever there is no risk for confusion. The term *deep* neural networks is used for networks with more than one hidden layer [12]. For such networks, the final output can be viewed as a composition of functions. For example, the output of a two-layer network can be described as:

$$y = f_2(W_2 f_1(W_1 x)) \tag{2.7}$$

where W_n and f_n denote the weight matrix and activation function of layer n, respectively. Note the allowance of using different types of activation functions in different layers.

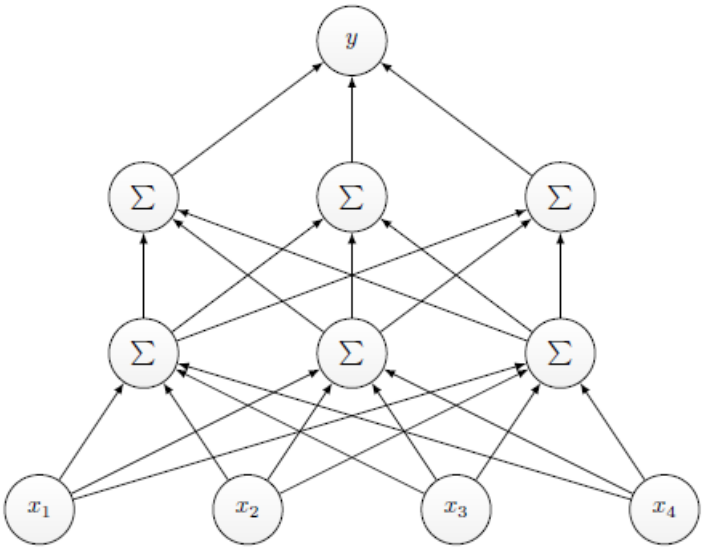

Figure 2.1: A neural network with four inputs x_1 to x_4, two hidden layers of three neurons each and one output y. Each hidden neuron, labeled with $\sum$, propagates $a = f(w^\top x)$ to the next layer. The final output y is computed in the same way.

The name *network* stems from the fact that the input x and output y of each layer can be broken down into components, commonly denoted as *neurons* [12]. This makes it possible to visualize the computational flow as a graph, where each node represents a neuron, as in Figure 2.1.

The intuitive role of each hidden layer is to learn some abstract feature in the input data, which is then propagated as an input to the next layer. A deep neural network, which consists of multiple layers, can thus be viewed as learning features in a dataset at different levels of abstraction [12]. Training a neural network to learn such features, and hence approximate a function $y = f^*(x)$, is a matter of finding appropriate weights W_n for each hidden layer. In this context, appropriateness is measured by a loss function $L(X, y) \in \mathbb{R}^+$ which describes how well f^* approximates the mapping between observed data samples $x_1, ..., x_n$ and their corresponding labels $y_1, ..., y_n$. For classification tasks, a common choice for the loss function is the categorical cross-entropy loss:

$$L(X, y) = -\sum_{i=1}^{N}\sum_{c=1}^{M} \mathbb{1}(y_i, c) \log(P(c \mid X_i)) \tag{2.8}$$

where N is the number of samples, M is the number of distinct classes, $\mathbb{1}(a, b)$ is the indicator function that returns 1 iff $a = b$ and 0 otherwise and $P(c \mid X_i)$ is the probability that sample X_i belongs to class c, according to the output of the network. This probability is usually computed by normalizing the final outputs of the network to values in the interval between 0 to 1. Note that $L(X, y)$ is always positive since the logarithm of a probability is always negative, which cancels with the leading negation. Thus, the goal is to get a loss close to zero, which is achieved by a set of weights that causes $P(y_i \mid X_i)$ to be close to 1. In practice, this is done by minimizing the loss function with an iterative gradient descent approach.

Algorithm 1 SGD

Require: Learning rate ϵ
Require: Initial weights W
Require: Some stopping criterion
  while stopping criterion is not met **do**
    Sample a batch of n samples from the training set: $X_i \in X,\ y_i \in y$
    Compute gradient estimate: $\hat{g} \leftarrow \frac{1}{n}\nabla_W L(X_i, y_i)$
    Update the weights: $W \leftarrow W - \epsilon\hat{g}$
  end while
  return W

Algorithm 1 illustrates the *Stochastic gradient descent* (SGD) algorithm, which is popular to train neural networks. In each iteration of the loop, a subset of training samples are chosen with which the gradient of the loss function is computed. The weights of the network are then updated by subtracting the estimated gradient multiplied by a prespecified learning rate.

2.2.1 Convolutional Neural Networks

A *Convolutional neural network* (CNN) is a type of neural network specialized for processing data with a spatial structure. It is particularly popular for visual data, which has an inherent 2-D structure to it [12]. The fundamental difference between a CNN and a FFNN, as described in the previous section, lies in the type of linear operation used in (2.3). Whereas FFNNs use the GEMM product Wx as input to the activation functions, CNNs employ another mathematical operation called *convolution*, which is typically denoted with an asterisk. Hence, we write the output of a convolutional layer as:

$$y = f(K * X) \tag{2.9}$$

where X is an input tensor and K is a *kernel* tensor (i.e. a weight tensor of the same size or smaller than X). Following our focus on visual data, we assume that K and X are three-dimensional tensors. In particular, we assume that the original input to the network is a tensor X of shape $w \times h \times c$ representing the width, height and the number of color channels of an image, respectively. In that case, the convolutional operation returns a tensor with the following entries:

$$R_{w,h} = (K * X)_{w,h} = \sum_x \sum_y \sum_z K_{x,y,z} X_{w+x,h+y,c+z}. \tag{2.10}$$

In the literature, R is usually called a *feature map* [12]. The convolutional operation can be thought of as sliding the kernel over the input, yielding a component-wise linear combination of the kernel and a patch of the input at each step. Figure 2.2 illustrates an application of the convolutional operation in 2-D.

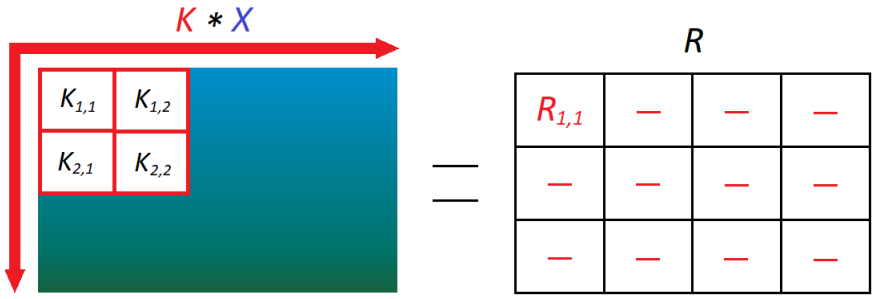

Figure 2.2: An input image X is convolved by a kernel K to produce a feature map R. The kernel can be thought of as sweeping through the input in both directions, producing $R_{w,h} = \sum_x \sum_y K_{x,y} X_{w+x,h+y}$ for each position w, h.

Note that it is possible, and even common, for a convolutional layer to contain several weight kernels. In such cases, each kernel will produce a distinct channel in the output feature map. Training a CNN is a matter of finding an appropriate set of weights for each kernel in the network. This is done in the same way as for FFNNs by defining a suitable loss function which is then minimized through a gradient descent approach [12].

The main rationale of using the convolutional operation instead of GEMM is that it leverages the idea of *sparse interactions* between the input and the weights. For example, in Figure 2.3, since the kernel K operates on a small patch of the image at a time, it can detect small, meaningful features such as edges of a particular shape or color in individual patches of the input image. GEMM, on the other hand, operates on the entire image at once which makes it more difficult to find such fine-grained features. A secondary reason for using convolutions is that it leverages *parameter sharing*. Again, referring to Figure 2.3, we see that the weight $K_{x,y}$ will operate on multiple pixels in X whereas GEMM would have multiplied each pixel with a distinct weight. In practice, this often leads to significant memory savings compared to FFNNs [12].

Similarly to hidden layers of an FFNN, each convolutional layer is typically viewed as learning features of successively higher levels of abstraction. Hence, in order for complicated patterns to be learned, the network needs to be deep. On the other hand, networks with too many layers may lead to unacceptable computational costs. Several architectural techniques have been proposed to allow deeper networks to be used while keeping the computational complexity within acceptable limits. One such technique, which is most prominently used in ResNet architectures [16], is the usage of residual layers, where the goal is to learn the residual of the input and a feature map, instead of the actual features themselves. In essence, this allows the layer to pass its input unchanged if it detects that it cannot learn any significant feature in the input [16]. Another technique is to use separable convolutions, in which the kernels of a convolutional layer are decomposed into a couple of smaller factors from which the original kernel can be reconstructed when needed [3]. MobileNet [30] is an example of a network with many separable convolutions.

2.3 Reinforcement Learning

Reinforcement learning is an example of an *unsupervised* machine learning technique. In contrast to supervised techniques, such as neural networks, there is no explicit supervision of the learner's performance. Instead, the learner has to actively explore the solution space and autonomously assess its own performance according to the outcome of its actions. As such, it is often a suitable approach for interactive problems where examples of good behaviour are not available. This section will give a light-weight introduction to the topic while focusing on the parts that are essential in order to understand the proposed solution to the research problem of this thesis. A comprehensive treatment of reinforcement learning, from which most of the material of this section is based on, is given by the book *Reinforcement Learning: An Introduction* by Richard S. Sutton and Andrew G. Barto [33].

Formally, the reinforcement learning problem is defined as an iterative interaction between an *agent* and an *environment*. At each discrete time-step $t \in \{0, 1, ..., T\}$, the agent observes a state s_t which describes the observable properties of the environment and a reward r_t which describes the immediate utility of being in that state. Given this information, the agent executes a new action a_t which transforms the environment, whereupon it observes a new state s_{t+1} and reward r_{t+1} (see Figure 2.3). The goal of the agent is to learn an optimal *policy* (a mapping from states to actions) by connecting experienced state-action pairs with their corresponding reward signals. The distinction between agent and environment is not always clear. For example, if the agent in question is a human, it is not obvious which part of the human should be included in the agent and environment, respectively. A general rule of thumb is that the environment should consist of all components of the problem that cannot be changed *arbitrarily*. Following this rule, the agent should merely be viewed as an abstract decision-making machine, leaving actuators such as arms and legs to the environment.

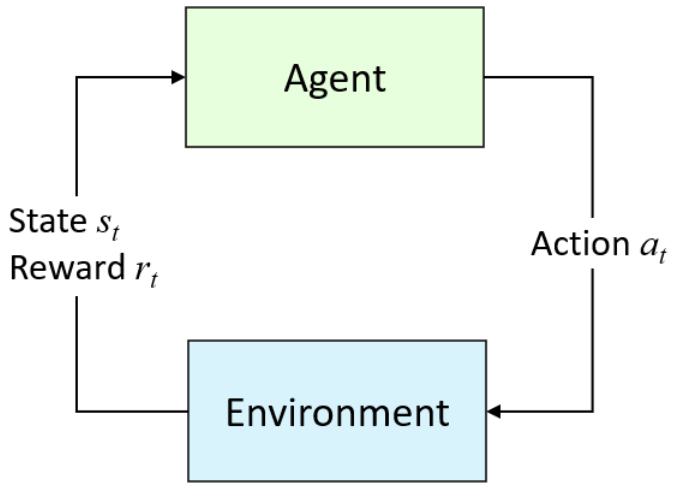

Figure 2.3: At time-step t, the agent receives a state-reward pair (s_t, r_t) and executes an action a_t.

It is usually recommended to design the reward function based on *what* to achieve rather than *how* to achieve it, especially for problems where there is no obvious heuristic for how an optimal solution can be obtained. Hence, a reward of $r_t \in \mathbb{R}^+$ should reflect that the agent is in a desirable state at time-step t. Similarly, a reward of $r_t \in \mathbb{R}^-$ should reflect that the agent is in an undesirable state at time-step t. However, it is important to note that the reward should reflect the *immediate* (short-sighted) utility of being in a particular state. The long-term utility of a particular state is denoted as the *return* and is commonly defined as:

$$G_t = \sum_{k=0}^{T} \gamma^k r_{t+k} \tag{2.11}$$

where γ is a discount factor, usually in the range (0,1], which can sometimes be utilized to discourage the agent to delay its reward accumulation. This parameter is also crucial for problems with no conventional endpoint (e.g. $T = \infty$) to prevent infinitely large returns.

Markov Decision Processes

Typically, the goal of the agent is to learn a policy that maximizes its expected return from a given start state. To achieve this, the agent must be able to determine a suitable action for each possible state it may visit. One dilemma that may arise in learning such a policy is that valuable information may be hidden in the trajectory of state transitions. This makes it much more difficult for the agent to select an action by looking at a single state in isolation. Ideally, future states should be independent of past states, given the information available in the present state. State spaces which possess this property are said to satisfy the Markov property, which is formally defined as:

$$P(s_{t+1} \mid s_t, a_t, ..., s_0, a_0) = P(s_{t+1} \mid s_t, a_t) \tag{2.12}$$

A reinforcement learning problem with a state space that satisfies the Markov property is commonly referred to as a *Markov decision process* (MDP). The dynamics of an MDP can be modeled succinctly by two functions. The transition function gives the probability that the agent ends up in a successor state s' after executing action a in state s:

$$T^a_{ss'} = P(s_{t+1} = s' \mid s_t = s, a_t = a) \tag{2.13}$$

The reward function gives the expected reward when executing an action a in state s which takes the agent to a successor state s':

$$R^a_s = E\left[r_{t+1} \mid s_t = s, a_t = a\right] \tag{2.14}$$

Note that the probabilistic nature of these functions allows for modelling stochastic environments where the exact dynamics are unknown. For such environments, Equations (2.13) and (2.14) can be estimated with, for example, maximimum likelihood estimation.

Learning Policies

Given an MDP, defined by the reward and transition functions, the goal of the agent is to learn an optimal policy. A policy is formally defined as a probability distribution $\pi(a \mid s)$ which assigns a probability to each action in a given state. We say that an agent follows a policy π if the agent samples its action from $\pi(a \mid s)$ for all states s. To define the notion of an optimal policy, we first need to define the value of a state and a state-action pair. The value of a state s, given that a policy π is being followed, is commonly defined as:

$$\begin{aligned} V_\pi(s) &= E_\pi \left[G_t \mid s_t = s \right] \\ &= E_\pi \left[\sum_{k=0}^{T} \gamma^k r_{t+k} \mid s_t = s \right] \\ &= \sum_{a \in A} \pi(a \mid s) \left(R_s^a + \gamma \sum_{s' \in S} T_{ss'}^a V_\pi(s') \right) \end{aligned} \tag{2.15}$$

Similarly, the value of being in a state s, executing an action a and then following a policy π is commonly defined as:

$$\begin{aligned} Q_\pi(s,a) &= E_\pi \left[G_t \mid s_t = s, a_t = a \right] \\ &= E_\pi \left[\sum_{k=0}^{T} \gamma^k r_{t+k} \mid s_t = s, a_t = a \right] \\ &= R_s^a + \gamma \sum_{s' \in S} T_{ss'}^a \left(\sum_{a' \in A} \pi(a' \mid s') Q_\pi(s', a') \right). \end{aligned} \tag{2.16}$$

If the dynamics of the MDP are known in advance, V_π and Q_π can be computed directly. If the dynamics are not known, they can be estimated by letting the agent explore the environment with a randomized policy. This technique is sometimes referred to as *warmup*. A policy π^* is said to be optimal iff:

$$\forall \pi, s \in S : \; V_{\pi^*}(s) \geq V_\pi(s). \tag{2.17}$$

The general procedure that is often used to find π^* is called *General policy iteration* (GPI) which iteratively performs two different steps:

1. **Policy Evaluation** - This step evaluates $V_\pi(s)$ and $Q_\pi(s,a)$ for all states s and state-action pairs (s,a), respectively.

2. **Policy Improvement** - This step derives a new policy π' that is greedy with respect to the newly estimated Q_π (e.g. $\forall s \in S : \; \pi'(s) = \arg\max_a Q_\pi(s,a)$).

These steps are executed in an iterative manner until some convergence criterion has been achieved; for example, if the updates of $V_\pi(s)$ are very small. There are many specializations of GPI; one that is particularly popular for problems with continuous action spaces (i.e. $a \in \mathbb{R}$) is the *Deep Deterministic policy gradient* (DDPG) [25] algorithm described below.

Algorithm 2 DDPG

1: Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\pi(s|\theta^\pi)$ with weights θ^Q and θ^π
2: Initialize target network Q' and π' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\pi'} \leftarrow \theta^\pi$
3: Initialize replay buffer R
4: **for** episode = 1, M **do**
5: Initialize random process $\mathcal{N}$ for action exploration
6: Receive initial obervation state s_1
7: **for** $t = 1, T$ **do**
8: Select action $a_t = \pi(s_t|\theta^\pi) + \mathcal{N}_t$
9: Execute action a_t and observe reward r_{t+1} and observe new state s_{t+1}
10: Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in R
11: Sample a random minibatch of N transitions $(s_i, a_i, r_{i+1}, s_{i+1})$ from R
12: Set $y_i = r_i + \gamma Q'(s_{i+1}, \pi'(s_{i+1}|\theta^{\pi'})|\theta^{Q'})$
13: Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
14: Update actor using the sampled policy gradient:

$$\nabla_{\theta^\pi} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\pi(s_i)} \nabla_{\theta^\pi} \pi(s|\theta^\pi)|_{s_i}$$

15: Update the target networks:

$$\theta^{Q'} \leftarrow \eta\theta^Q + (1-\eta)\theta^{Q'}$$
$$\theta^{\pi'} \leftarrow \eta\theta^\pi + (1-\eta)\theta^{\pi'}$$

16: **end for**
17: **end for**

The DDPG algorithm uses two neural networks, called the actor and the critic, to represent the current policy and the state-action values. It also uses two target networks to facilitate smooth updates of the actor and critic networks. The architecture of these networks is specified by the application. GPI is performed in lines 4-15, but instead of checking for convergence, it performs a fixed number of policy iterations (episodes). DDPG is an *off-policy* algorithm, which means that the network updates are based on a different policy than the one followed by the agent. Specifically, as shown in line 8, the agent follows policy π with some dynamic noise added to facilitate exploration. The observed transitions are stored in a replay buffer R, as shown in line 10. The network updates are then based on a random minibatch of N transitions, sampled from R. Note that these N transitions could have been experienced through a completely different policy, which causes the off-policy learning behaviour. The actor-critic updates in lines 13-14 can be done with an arbitrary network learning algorithm, such as SGD. The target networks, however, are updated by modifying the weights directly at a rate of $\eta \in [0, 1)$. After M episodes, a solution to the MDP is given by $\pi(s|\theta^\pi)$.

3
Context

This chapter introduces recent work in the field of network optimization, including neural architecture search, knowledge distillation and network compression.

3.1 Neural Architecture Search

Neural architecture search refers to the usage of optimization techniques to design a suitable neural architecture for a given machine learning task. Much of the recent work in this area stems from the reinforcement learning framework proposed by Zoph and Le [34]. In this framework, an architecture is represented as a sequence of tokens, each symbolizing a building block of the architecture. A RNN controller is used to generate candidate sequences, from which models are built, trained and validated to obtain an accuracy score. Given this score, a reinforcement learning procedure is used to update the controller into generating better candidates. The authors showed that this approach was able to design both CNN- and RNN-architectures with better predictive performance than previous state-of-the-art architectures for vision and language tasks, respectively. While the most common objective is to find an architecture that, once trained, obtains a higher predictive performance than handcrafted ones, recent work have explored the inclusion of other criteria, such as inference latency and memory footprint, into the scoring function [34].

3.2 Knowledge Distillation

In large-scale machine learning, the requirements of a model will typically change throughout its lifecycle. During the training stage, the model must be able to extract complex patterns in high-dimensional datasets. Resource consumption is usually not a concern in this phase, since large computational resources will typically be available. Once the model is deployed, however, it is often required to operate in more resource constrained environments with stringent real-time requirements.

Knowledge distillation attempts to solve these conflicting requirements by transferring (or distilling) the patterns learned by a large "teacher" model to a smaller "student" model. One method to achieve this kind of distillation was proposed by Hinton et al. [19], in which the student network is trained to mimic the teacher network by means of outputting the same logits (the inputs to the final softmax layer) as the teacher on a held-out transfer set. The rationale of using the teacher's logits as the training target is that it provides a deeper insight into the feature space

than the discrete labels. For example, an output distribution of [0.5, 0,49, 0.01] does not just suggest that the example should be classified as c0, but it also suggests that that c0 is more distinguishable from c2 than it is from c1. In this sense, the logit distribution can be viewed as a compact representation of the features extracted by the teacher.

One obstacle with knowledge distillation is to find an appropriate architecture for the student network that performs satisfactory, in terms of both predictive and computational performance. Ashok et al. [1] proposed a reinforcement learning algorithm that starts with the teacher network and constructs a sequence of smaller, student networks, by performing architecture altering operations such as layer removal. At each timestep, one such operation is performed on the current student to create a new, smaller student. Knowledge distillation is then performed on the new student and its performance is used as a reward signal to the reinforcement learning procedure. Experiments conducted on various VGG and ResNet architectures showed that this approach is capable of reducing the number of parameters by 3x (on ResNet-18) to 127x (on VGG-13).

3.3 Network Compression

Prior work in the area of network compression can be divided into three categories: pruning, quantization and factorization. This section gives an overview of recent work in each of these categories, consecutively.

3.3.1 Pruning

The rationale behind pruning is that low-weight connections cause small activations and can thus be removed without disturbing the activation patterns. To exploit this insight, Han et al. [15] proposed an algorithm that removes all connections below a prespecified threshold by setting the corresponding weights to zero. The remaining weights are then fine-tuned with a lower learning rate. These two steps are repeated in an iterative fashion until convergence. Finally, the resulting weights are stored in a sparse matrix format to avoid storing the zero-valued entries. Evaluations on various network architectures, including LeNet-300 [24], AlexNet [23] and VGG-16 [32], showed that this approach was able to reduce the number of weights by 9× to 12× and the number of floating-point operations by 3× to 12×, with no drop in predictive performance.

One limitation of such *fine-grained* pruning is that the resulting weight tensors contain irregular sparsity patterns. This makes the compressed networks difficult to accelerate on conventional hardware. Hardware specialized for accelerating sparse tensor operations, such as the *Efficient inference engine* (EIE) [13], have been proposed but are not widely available in everyday devices. As a consequence, the compression rates obtained by fine-grained pruning in experimental settings are difficult to achieve in practice.

Due to the inherent limitation of fine-grained pruning, much of the recent work has focused on *coarse-grained* pruning, in which larger blocks of weights are considered for removal. He et al. [18] proposed an algorithm that removes entire filters from a CNN layer. Since the removal of a filter reduces the number of output channels, they call this approach *channel pruning*. Specifically, given a feature map and a target sparsity α, their algorithm first selects the most representative feature channels through LASSO regression such that a sparsity ratio of α is achieved. The channels that were not selected are then removed, along with the filters that produced those channels and the corresponding filter channels in the next convolutional layer. Finally, the weights of the next feature map are reconstructed using a linear regression approach on the remaining feature channels. This approach was able to speed up VGG-16, ResNet-50 [16] and Xception [3], all pre-trained on ImageNet, by a factor of 2× with no loss of predictive performance.

Following their original work on channel pruning, He et al. [17] leveraged reinforcement learning to learn the optimal sparsity ratio for each layer of a convolutional network. In their approach, which they call *AutoML for Model Compression* (AMC), a *Deep deterministic policy gradient* (DDPG) [25] agent is employed to explore an environment in which each state represents a convolutional layer and its computational cost. Given a state s, the action space is defined as $\pi(s) \in [0, 1]$, which corresponds to the target sparsity of the layer associated with s. Standard channel pruning is then applied to reach the target sparsity. Experiments conducted on VGG-16, ResNet-50 and MobileNet showed that the policy found by the agent outperforms handcrafted heuristics, allowing for slightly larger compression rates without reliance on domain expertise.

Liu et al. [26] proposed an alternative, meta learning approach to reconstruct the weights after each filter removal. In their approach, an auxiliary network is trained to generate the weights of a pruned network. Evolutionary search is then used to find the optimal network structure (i.e. number of channels per layer), where each candidate structure is fitted with the weights generated by the auxiliary network. A candidate network is evaluated on validation data, after which crossover and mutation is applied to generate another set of candidates. On ResNet-50 and MobileNet, this approach was shown to achieve larger FLOP reductions with slightly larger accuracy, compared to AMC. The obvious drawback of this approach is the reliance on an auxiliary network, which makes it difficult to achieve dynamic compression under resource constrained settings.

3.3.2 Quantization

Quantization methods aim to reduce the bitwidth precision of the weights in a neural network. A simple approach to do this is to manually alter the datatype of each weight (e.g. from 32 bit float to 16 bit int) [22]. There are also software development tools, such as TensorRT [37], which can do this kind of transformation automatically. A more sophisticated approach was proposed by Han et al. [14], in which k-means

clustering is used to organize the weights into different clusters. Each weight is then replaced with the centroid index of its assigned cluster. Finally, the shared weights are fine-tuned with a standard optimization algorithm, such as stochastic gradient descent [12]. Results showed that 32 clusters were enough to quantize the weights of LeNet-300, AlexNet and VGG-16 without losing predictive performance. This allowed each weight to be stored using 5 bits instead of 32.

Wang et al. [38] acknowledged that the optimal number of clusters to use in the previously discussed quantization approach can vary between different hardware architectures. For example, the NVIDIA Turing GPU architecture supports 1-bit, 4-bit, 8-bit and 16-bit arithmetic operations, while other architectures provide less flexibility. Furthermore, they found that the optimal number of clusters can vary between network layers. To overcome this issue, they introduced *Hardware aware quantization* (HAQ), in which a DDPG agent is trained to find the optimal quantization policy (i.e. number of clusters for each layer) in a similar way as AMC. More specifically, for each layer in a network, the agent gets to pick an integer action b, which corresponds to the number of bits with which to quantize the layer. The layer is then quantized with 2^b clusters using the method proposed by Han et al. [14] Once the compression is finished, the network is fune-tuned for one epoch on the entire training set. Experiments conducted on the BitFusion [31] and Bit-Serial Matrix Multiplication Overlay (BISMO) [36] accelerators showed that this approach can reduce the latency and energy consumption of MobileNet by 2× with negligible accuracy loss.

Previously mentioned approaches are commonly referred to as *post-training* quantization methods. Meanwhile, quantization can also be incorporated into the training phase of a network. Hubara et al. [20] proposed a training scheme in which the weights are forced to attain values in the set $\{0, 1\}$. This scheme did not only result in a more efficient training, it also reduced inference latency by 7× without suffering any accuracy loss compared to a similar network architecture with floating-point weights. Chen et al. [2] proposed an alternative training scheme in which parameters are randomly organized into groups according to a hash function. The parameters in each group are then forced to share the same weight. This method can lead to a very small memory footprint since the weight for a given parameter can be obtained dynamically from the hash function when needed. On the MNIST dataset [7], this training method was able to compress the size of a five layer network by 32×, with no significant accuracy loss compared to a normally trained network of the same architecture.

3.3.3 Factorization

Attempts have been made to compress neural networks using matrix factorization techniques. Denton et al. [9] showed that *Singular value decomposition* (SVD) can be used to accelerate fully-connected layers by up to 13× with negligible loss of predictive performance. They also proposed *Biclustering approximation*, which first uses clustering to organize the weights into different groups according to their values. SVD is then applied to factorize each cluster, separately. In experiments conducted on a 15 layer CNN pre-trained on the ImageNet dataset, this approach was shown to outperform regular SVD on the convolutional layers, providing accelerations of up to 3× while reducing their sizes by up to 5×. However, it did not outperform the regular SVD approach on the later, fully-connected layers.

Dubey et al. [11] considered another approach based on *coreset extraction*, which refers to the general idea of approximating a large set of points with a smaller set, which does not necessarily need to be part of the original set, while preserving some desirable property of the original set. In the context of network compression, the point sets to approximate are the weight matrices and the property to preserve is the activation patterns (i.e. the matrix multiplications). To do this, they used an algorithm known as *Sparse principal component analysis* (SPCA), which incorporates sparsity constraints into the standard PCA objective. They also extended this algorithm with an activation-weighted importance score for each convolutional filter, allowing the algorithm to provide greater compression of unimportant filters. This approach was able to reduce the size of AlexNet by around 10× with no loss of predictive performance. This compression rate was achieved without fine-tuning the model, which allows for a fast and simple implementation.

3.3.4 Hybrid Approaches

Some work has been made to incorporate different types of compression techniques in a single pipeline. In the work by Han et al. [14], the authors proposed *Deep compression*, a pipeline consisting of pruning, quantization and Huffman coding. They showed that this pipeline was capable of reducing the size of VGG-16 by 49×. One limitation of this method is that fine-tuning is used after the pruning and quantization stages, which prohibits fast, dynamic compression in resource-constrained runtime environments. In the work by Dubey et al. [11], a pipeline of pruning, coreset extraction and Huffman coding was used to reduce the memory footpring of AlexNet by 832×, as well as its inference latency by 2×, with no significant loss of predictive performance. In contrast to Deep compression, these compression rates were achieved without fine-tuning. These two pieces of work show that different compression techniques may synergize well with each other, allowing for larger compression rates than either of the individual components in isolation.

4
Methods

This chapter describes the approach used to solve the underlying problem of this thesis. Since the approach is largely based on explicit hardware feedback, we start with a full description of the performance metrics used to evaluate a neural network, as well as the profiling methods used to measure those metrics. After that, we propose a novel optimization formulation for compressing neural networks that are part of a safety-critical real-time system. We then propose a solution to this optimization problem which consists of a reinforcement learning framework. We will see that this general framework can be specialized into three concrete algorithms by plugging in different types of compression actions. Finally, we give a description of the methods used to evaluate the proposed solution, including its three specializations.

4.1 Profiling

One of the main novelties of this work is the explicit usage of direct hardware metrics in the optimization and evaluation processes. As we are not aware of any existing tool for profiling neural networks according to the metrics of interest, we present such a tool as part of this work. The profiler takes a pre-trained network and a dataset, and evaluates the network according to 12 performance metrics, which can be grouped into three classes:

1. **Predictive performance metrics**: These metrics capture the modeling accuracy and flexibility of a network. In our case, we use the top-1 and top-5 classification errors.

2. **Indirect performance metrics**: These metrics capture the resource consumption of a network in an abstract way. In our case, we use the number of parameters and the number of floating point operations of a network.

3. **Direct performance metrics**: These metrics capture the actual resource consumption of a network, as measured directly from the target hardware using a combination of `nvprof` [5] and `tegrastats` [6] utilities. The direct performance metrics considered in this work are effect, energy, initialization time, loading time, inference latency, throughput and the number of floating point operations carried out per second.

A full description of the performance metrics is given below.

- **Top-1 error**: This metric is computed as $\frac{error}{total}$ where *error* denotes the number of misclassifications and *total* denotes the number of test samples.

- **Top-5 error**: This metric is computed in a similar way as the top-1 error, with the difference being that the five most probable classes are treated as an aggregate prediction. More precisely, $error = \sum_k \min_i d(c_i, C_k)$ where k denotes the number of samples, C_k denotes the correct class of sample k, c_i denotes the class with the i'th highest probability in the classifier's output, $i \in [1, 5]$ and $d(a, b)$ is 0 if $a = b$ and 1 otherwise.

- **Parameters**: The number of weights and biases of a target network, measured in millions.

- **Model size**: The memory footprint of the model, measured in megabytes.

- **FLOP(s)**: The number of floating point operations required to propagate a single image through the target network, measured in billions. MAC operations are counted separately. Furthermore, vectorized (e.g. SIMD) operations are counted component-wise.

- **FLOP/s**: The number of floating point operations per second performed by the hardware when propagating batches of 32 images each through the target network. As with FLOP(s), MAC operations are counted separately and vectorized operations are counted component-wise. It is evaluated in the scale of billions per second.

- **Throughput**: The number of images that can be channeled through the target network in one second of wall clock time. Note that this quantity is not necessarily derivable from FLOP(s) and FLOP/s, because memory transfers are not taken into account in those computations.

- **Latency**: The time it takes to propagate a single image through the target network, measured in wall clock milliseconds. Host to device and device to host loading times are excluded from this quantity as they can obscure the actual speedups for fast networks.

- **Loading time**: The time it takes to move the network (i.e. parameters and instruction set) to the computational device, measure in milliseconds.

- **Initialization time**: The time it takes to initialize the network from scratch.

- **Energy**: The amount of energy consumed by propagating a batch of 32 images through the target network, measured in joules.

- **Effect**: The max effect used when propagating seven batches of 32 images each through the target network, measured in watts.

4.2 Optimization

A proper formulation of an optimization problem consists of two parts: an objective function, which specifies a quantity to optimize, and a system of constraints that has to be achieved by a feasible solution. For network optimization in general, the objective function is typically not problematic as its formulation allows for a great degree of freedom. The same thing cannot be said for the system of constraints. One of the main obstacles with network compression is that the feasibly obtainable trade-offs between predictive and runtime performance is not known in advance. As such, it is seemingly impossible to guarantee the feasibility of a system consisting of both resource *and* accuracy constraints. This also makes it difficult to evaluate the quality of compression algorithms. However, for safety-critical real-time systems, it is necessary to enforce both types of constraints.

He et al. [17] proposed to split the problem into two distinct optimization protocols. In the *accuracy-guaranteed* protocol, the objective is to minimize the resource cost of the network while forcing the accuracy above a specified threshold. In the *resource-constrained* protocol, the objective is to maximize predictive performance while forcing the resource costs below a specified budget. In their work, they used the number of parameters as the sole resource metric, and classification accuracy as the sole metric of predictive performance. In our work, we extend their definition to allow multiple resource constraints in both the objective function of the accuracy-guaranteed protocol and in the constraints section of the resource-constrained protocol. The main rationale of this partitioning is that each protocol instance is guaranteed to have a trivial feasible solution. For the accuracy-guaranteed protocol, the trivial solution is obtained by leaving the original network intact. For the resource-constrained protocol, it is obtained by deleting the entire network.

The only downside of this partitioning is that it does not allow for compressing a network that has both accuracy and resource constraints, which may appear in safety-critical real-time systems. As was noted in Section 1.3, however, this work is explicitly targeted towards systems which consists of multiple networks, not all of which are safety-critical at all times. We claim that the proposed way of partitioning the problem into these two protocols fits those systems well since a network can be compressed with different approaches at runtime, depending on whether it is currently safety-critical or not. With this demarcation in mind, there are not many conceivable use cases for including both accuracy and resource constraints in the objective function either, which results in a clean formulation of the objective function in both protocols.

4.2.1 Accuracy-guaranteed Optimization

The intended use case for the accuracy-guaranteed protocol is to compress a network that is currently safety-critical. Since the network is assumed to operate within a real-time system, it is also desirable to minimize its resource costs. Hence, we formulate this protocol as a minimization (of resource costs) problem subject to a hard accuracy constraint as follows:

$$\begin{aligned} \min \quad & \mathbf{w}^\top \mathbf{r} \\ \text{s.t.} \quad & p \geq c \end{aligned} \tag{4.1}$$

where $\mathbf{r}$ is a vector of resource costs and p is a measurement of the predictive performance of the model. This formulation is general enough to allow for a wide variety of resource and accuracy metrics and the linear objective function allows for emphasizing certain resources more than other.

4.2.2 Resource-constrained Optimization

The intended use case for the resource-constrained protocol is to compress a network that is currently *not* safety-critical. Instead, since the network is assumed to operate within a real-time system, it is of utmost importance to clamp its resource consumption below a specified budget. However, as long as the resource budget is met, there is usually no need to compress the network further. Hence, we formulate this protocol as a maximization (of predictive performance) problem subject to a system of resource constraints as follows:

$$\begin{aligned} \max \quad & p \\ \text{s.t.} \quad & \forall i : r_i \leq c_i \end{aligned} \tag{4.2}$$

where p is a measurement of the predictive performance of the model, $\{r_1, ..., r_n\}$ are the computational resources under consideration and $c_i \in \{c_1, ..., c_n\}$ is the constraint for resource r_i. As with the accuracy-guaranteed protocol, this formulation is general enough to support different types of resource constraints which is achieved by plugging in different values for r_i and c_i.

4.3 Algorithms

Inspired by the work of He et al. [17] and Wang et al. [38], we propose to solve the problem stated in 4.2 with reinforcement learning. On a high level, we consider an agent that goes through the network one layer at a time. At each time step, the agent receives a state embedding consisting of layer dimensions and network performance statistics. Given this state embedding, the agent gets to pick an action $a \in [0, 1)$ which represents a reduction ratio with which to compress the current layer. Once the final layer is compressed, a reward is computed based on the achieved compression rates in the light of the protocol instance. The agent then uses this reward signal to update its policy. Figure 4.1 gives a schematic overview of the optimization loop for the proposed compression solution.

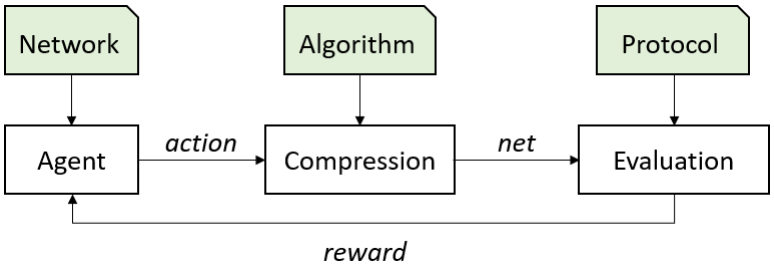

Figure 4.1: Overview of the optimization loop.

What follows next is a detailed description of the components of the reinforcement learning framework: the agent, the state representation, the compression actions and the reward functions.

4.3.1 Agent

Since we consider a continuous action space, we use a DDPG agent to learn an optimal compression policy. The details of this agent are described in Section 2.3. The actor and critic networks have the same architecture: a FFNN with two hidden layers of 400 and 300 neurons, respectively. The learning rates were set to 10^{-4} for the actor network and 10^{-3} for the critic network.

4.3.2 State

At each time step, the agent receives a state embedding which consists of the following components:

1. *id* denotes the layer index.
2. *type* denotes the layer type (0 for convolutional and 1 for linear layers.)
3. *out* denotes the number of output channels of the layer.
4. *in* denotes the number input channels of the layer.
5. h denotes the height of the input feature map.
6. w denotes the width of the input feature map.
7. *stride* denotes the stride of the kernel (0 for linear layers.)
8. k denotes the side length of the kernel (1 for linear layers.)
9. *vol* denotes the volume of the weight tensor.
10. *macs* denotes the number of macs in the layer.
11. *prev* denotes the previous action.
12. *rest* denotes the number of macs in following layers.
13. *rem* denotes the fraction of macs that remains in the entire network.

Note that we do not include direct resource metrics, such as latency and throughput, in the state embedding. We discovered that the agent is perfectly capable of finding the correspondence between macs and the desired compression criteria. In fact, the vastly increased state space resulting in including such components caused even longer convergence times. Secondarily, it is a technical difficulty to accurately estimate the consumption of certain resources for a particular layer.

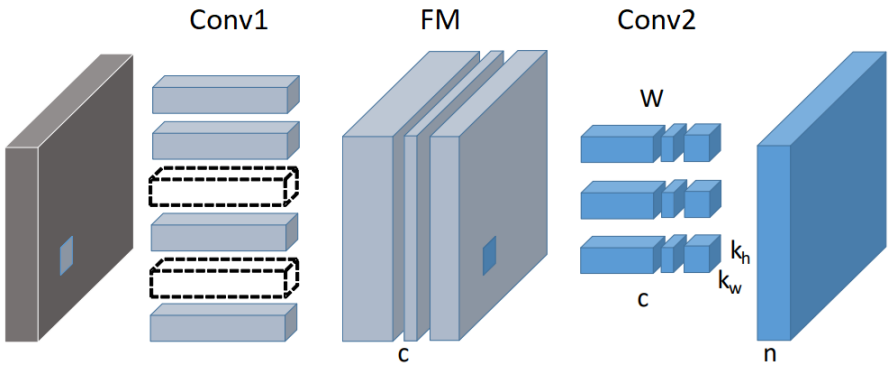

Figure 4.2: Illustration of pruning. Two kernels are removed from Conv1 which reduces the number of channels in FM. The corresponding channels are removed from every kernel in Conv2.

4.3.3 Actions

In the most general sense, the action space consists of a single continuous action $a \in [0, 1)$ which represents the target compression rate for the current layer. This value is then fed as an input to a particular algorithm with which the layer is compressed. In our experiments, we consider three types of compression actions: channel pruning, filter pruning and Tucker decomposition.

Filter Pruning

With filter pruning, a is interpreted as the fraction of filters to keep in a convolutional layer. Conversely, $1-a$ is interpreted as the fraction of filters to remove. For a layer l_i with f filters, we remove the $k = \lfloor f \times (1-a) \rfloor$ filters with the lowest L_1-rank. Note that this reduces the number of channels in the feature maps produced by l_i. Hence, we must also remove the corresponding k channels from every filter in layer l_{i+1}. Because of this, filter pruning can only be applied on pairs of convolutional layers. See Figure 4.1 for an illustration of this action type.

Channel Pruning

While filter pruning looks at the kernels of a convolutional layer, channel pruning looks at the channels of an average of presampled feature maps from that layer. Specifically, the $k = \lfloor f \times (1-a) \rfloor$ channels with the lowest L_1-rank are removed, along with the kernels in the layer that produced that feature map and the corresponding channels in the next layer. One advantage of having representative samples of feature maps is that it enables a simple approach to feature map reconstruction. When pruning a layer L, its weight tensor W is transformed into a smaller tensor W'. Similarly, since kernels are removed from L_{i-1}, the feature map X that goes into L is transformed into a smaller feature map X'. When convolving X' with W', we get a new output Y' which may be different from the original output Y.

With feature map reconstruction, the goal is to alter W' such that $Y' = W' * X' \approx Y = W * X$. Assuming that the layer uses the relu activation function, which is the most common activation function in modern CNNs, we state the reconstruction problem as a minimization problem:

$$W' = \arg\min_{W'} ||Y - Y'||_F^2 \tag{4.3}$$

which can be solved in closed form with least-squares regression.

Tucker Decomposition

Tucker decomposition is a tensor factorization method and is commonly viewed as a higher-order variant of *Singular value decomposition* (SVD). Unlike filter and channel pruning, this action can be applied on single convolutional layers. To see how this is done, we assume a convolutional weight tensor $W \in \mathbb{R}^{w \times h \times i \times o}$ where w is the kernel width, h is the kernel height, i is the number of input channels and o is the number of output channels. Tucker decomposition factorizes W, along a subset of dimensions, into a core tensor and several matrix factors. For the case of convolutional weight tensors, w and h are usually very small. Because of this, we perform the decomposition along the third (number of input channels) and fourth (number of output channels) dimensions.

The action a that is generated by the agent is treated as a fractional reduction in the rank of the third and fourth dimensions of W. In particular, we derive the new ranks for these dimensions, $R3$ and $R4$, as follows:

$$\begin{aligned} R_3 &= \lceil a \times I \rceil \\ R_4 &= \lceil a \times O \rceil \end{aligned} \tag{4.4}$$

The goal is then to decompose W into a core tensor $X \in \mathbb{R}^{w \times h \times R_3 \times R_4}$ and two matrix factors $U \in \mathbb{R}^{i \times R_3}$ and $V \in \mathbb{R}^{o \times R_4}$ in such a way that the full weight tensor can be reconstructed with minimal loss. Formally, the entries of the reconstructed weight tensor W' are defined as:

$$W'_{i,j,i,o} = \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} W_{i,j,r3,r4} \times U_{i,r_3} \times V_{o,r_4} \tag{4.5}$$

which lets us formulate the minimal reconstruction error problem as:

$$W' = \arg\min_{W'} ||W - W'|| \tag{4.6}$$

Using the method of *Higher order singular value decomposition* (HOSVD), we approximate a solution to REF by:

$$\begin{aligned} &U = R_3 \text{ left leading singular vectors of } W_3 \\ &V = R_4 \text{ left leading singular vectors of } W_4 \\ &X = W \times_3 U \times_4 V \end{aligned} \tag{4.7}$$

where $\times_n$ denotes the n-mode tensor product. For a more comprehensive treatment of tensor algebra and decompositions, the reader is referred to [21].

4.3.4 Rewards

The reward is computed by first evaluating the network on a held-out set of 250 samples using the profiling tool described in 4.1. The metrics of interest are then channeled into a reward function. As stated in Section 4.2, we consider two types of optimization protocols: the accuracy-guaranteed protocol and the resource-constrained protocol, which naturally results in two distinct reward functions.

Accuracy-guaranteed Optimization

The objective of the accuracy-guaranteed scenario is to minimize the cost across different computational resources given that the accuracy is above a certain threshold. In our experiments, we set this threshold to be 10% lower than the accuracy of the original model. If this constraint is met, the reward given to the agent is:

$$r = -(w_1 \times \text{frac}(lat) + w_2 \times \text{frac}(j) + w_3 \times \text{frac}(fps) + w_4 \times \text{frac}(size)) \tag{4.8}$$

where frac(r) denotes the fractional improvement of resource r, lat is the inference latency, j is the energy consumption, fps is the throughput, and $size$ is the model size. Refer to Section 4.1 for a complete description of these metrics. In this definition, w_i is a user-defined weight for resource i that can be specified by the user to put a greater emphasis on some resources than other. In our experiments, we use $\forall i : w_i = 0.25$ to give an equal emphasis to each of the four different resources. If the constraint is not met, the agent is given a flat reward of -50. It was empirically found that this number is large enough that the agent should always prioritize to keep the accuracy above the specified threshold. On the other hand, it is not large enough to incentivize the agent to refrain from performing any compression at all.

Resource-constrained Optimization

The objective of the resource-constrained optimization is to maximize model accuracy given that a system of resource constraints is satisfied. If the constraints are met, the reward given is simply:

$$r = acc \tag{4.9}$$

where acc is a measurement of the predictive performance of the model. In our experiments, we choose the top-1 error as the predictive performance metric. Similarly to the accuracy-guaranteed reward function, the agent is given a flat reward of -50 whenever the constraints are not satisfied. Since $acc \geq 0$, this should incentivize the agent to focus on the resource constraints first. In our experiments, we define the resource constraints as:

$$\begin{aligned} size &\leq 0.8 \times \text{original size} \\ latency &\leq 0.8 \times \text{original latency} \end{aligned} \tag{4.10}$$

4.4 Evaluation

To evaluate the compression methods described in 4.3, we used three publicly available networks of different architectures that are commonly used for vision tasks in real-time systems: ResNet-50 [16], ResNet-18 [16] and MobileNetV2 [30], all pretrained for image classification on ImageNet [29]. Validating each candidate network produced by the compression algorithms on the entire ImageNet dataset was not feasible due to time constraints. Instead, we used a subset of ImageNet that was constructed by including all test images from 20 randomly sampled classes. The sampled classes are listed in Table 5.1. From this subset, 250 images were used for training (i.e. to compute the episode reward for the reinforcement learning agent) and the remaining 750 images were held out for evaluation of the compressed networks. Note that this partitioning leaves room for slight discrepancies between the accuracy the error that the agent can see during optimization and the error obtained in the final evaluations.

Id	Description	Id	Description
n01440764	Tench	n01530575	Brambling
n01443537	Goldfish	n01532829	House finch
n01484850	White shark	n01534433	Snowbird
n01491361	Tiger shark	n01537544	Indigo bird
n01494475	Hammerhead shark	n01558993	American robin
n01496331	Electric ray	n01560419	Bulbul
n01498041	Stingray	n01580077	Jay
n01514668	Rooster	n01582220	Magpie
n01514859	Hen	n01592084	Chickadee
n01518878	Ostrich	n01531178	Goldfinch

Table 4.1: The randomly sampled ImageNet classes.

Target Hardware

All experiments were executed on the Nvidia Jetson AGX Xavier Developer Kit [4]. This hardware was selected as it closely resembles the hardware used in ADAS systems, which gives reliable estimates of direct performance metrics for such systems. The specifications for this device are given in Table 4.3.

GPU	512-core Volta GPU with Tensor Cores
CPU	8-core ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3
Memory	32GB 256-bit LPDDR4x | 137GB/s
Storage	32GB eMMC 5.1
DL Accelerator	(2x) NVDLA Engines
Vision Accelerator	7-way VLIW Vision Processor
Encoder/Decoder	(2x) 4Kp60 | HEVC/(2x) 4Kp60 | 12-bit support

Table 4.2: Specifications of the Nvidia Jetson AGX Xaview Developer Kit.

5
Results

In this chapter, we examine the compression rates achieved by the reinforcement learning agent when using channel pruning, filter pruning and Tucker decomposition actions, respectively. The agent's performance on the accuracy-guaranteed and resource-constrained protocols are presented in 5.1 and 5.2. Finally, in Section 5.3, we compare these results with those obtained by the TensorRT inference optimizer.

5.1 Accuracy-guaranteed Optimization

In this experiment, we compare the compression actions according to the accuracy-guaranteed protocol. For each type of action, the agent is first warmed up with 50 episodes of random exploration and then trained for 150 episodes of exploitation. To compensate for slow convergence rates, the action space was restricted to $a \in [0.3, 1]$ for each action type.

Table 5.1 displays the compression rates achieved on ResNet-50, where the highest reward of -0.78 was achieved with channel pruning. With this policy, the top-1 error was increased by 4.45 which is well within the 10% loss region posed by the accuracy constraint. The highest compression rate was obtained for the loading time, which was reduced by 35%, and the lowest was obtained for energy consumption and throughput, which were reduced by 17%. Note that, even though the size of the network was reduced by 33%, its initialization time was increased by 84%. The second highest reward of -0.94 was achieved with filter pruning. While this reward suggests that the top-1 error was within the acceptable loss region on the training set, the policy did not generalize well enough to the test set where the top-1 error was increased by 18.51. Furthermore, the compression rates were not as high as those obtained by channel pruning. With Tucker decomposition, the agent learned a *no operation* (NoOp) policy (i.e. selecting $a = 1$ for every layer) and kept the reference network intact.

Table 5.2 displays the compression rates achieved on ResNet-18. The highest reward of -0.83 was achieved with channel pruning. With this policy, the top-1 error was increased by 2.8, which is well within the acceptable loss region. The highest compression rate was obtained for inference latency, which was reduced by 33%, and the lowest was obtained for energy consumption, which was reduced by 5%. As with ResNet-50, the initialization time was increased, even though the size of the network was reduced. Filter pruning and Tucker decomposition yielded NoOp policies.

Metric	Unit	Ref	CP	FP	TD
Top-1 Error	(%)	**5.19**	9.64	23.7	**5.19**
Top-5 Error	(%)	**2.23**	3.57	7.15	**2.23**
Model size	(MB)	102	**66.7**	93.0	102
Parameters	(M)	25.5	**16.6**	23.2	25.5
FLOP(s)	(G)	6.45	**4.35**	6.24	6.45
FLOP/s	(G/s)	**447**	349	404	441
Throughput	(FPS)	66.4	**80.2**	64.7	65.9
Latency	(ms)	33.8	**26.7**	27.6	33.4
Loading time	(ms)	24.7	**16.3**	21.0	24.6
Init. time	(s)	**13**	24	21	**13**
Energy	(J)	0.35	**0.29**	0.33	0.35
Effect	(W)	4.32	**3.54**	4.62	4.32
Reward	-	-1.00	**-0.78**	-0.94	-1.03

Table 5.1: Algorithm comparison on ResNet50. The ref column describes the properties of the original network and the remaining columns describe the properties of the network when compressed with different algorithms: channel pruning (CP), filter pruning (FP) and Tucker decomposition (TD).

Metric	Unit	Ref	CP	FP	TD
Top-1 Error	(%)	**20.5**	23.2	**20.5**	**20.5**
Top-5 Error	(%)	**4.5**	5.8	**4.5**	**4.5**
Model size	(MB)	48.8	**40.7**	48.8	48.8
Parameters	(M)	11.7	**10.2**	11.7	11.7
FLOP(s)	(G)	1.82	**1.65**	1.82	1.82
FLOP/s	(G/s)	291	292	**294**	291
Throughput	(FPS)	153	**180**	155	152
Latency	(ms)	21.0	**13.9**	20.7	20.9
Loading time	(ms)	11.93	**10.54**	11.89	11.90
Init. time	(s)	**12**	22	**12**	**12**
Energy	(J)	0.19	**0.18**	**0.18**	0.19
Effect	(W)	2.01	**1.85**	2.17	2.06
Reward	-	-1.00	**-0.83**	-0.98	-0.99

Table 5.2: Algorithm comparison on ResNet18. The ref column describes the properties of the original network and the remaining columns describe the properties of the network when compressed with different algorithms: channel pruning (CP), filter pruning (FP) and Tucker decomposition (TD).

Metric	Unit	Ref	CP	FP	TD
Top-1 Error	(%)	**18.8**	19.6	62.1	**18.8**
Top-5 Error	(%)	**4.02**	4.46	44.6	**4.02**
Model size	(MB)	14.2	**11.7**	13.3	14.2
Parameters	(M)	3.50	**2.87**	3.28	3.5
FLOP(s)	(G)	0.72	**0.58**	0.68	0.70
FLOP/s	(G/s)	128	116	**132**	125
Throughput	(FPS)	176	**200**	193	177
Latency	(ms)	14.0	**12.0**	13.1	13.9
Loading time	(ms)	3.14	**2.72**	2.96	3.11
Init. time	(s)	**12**	17	17	**12**
Energy	(J)	0.18	**0.16**	**0.16**	1.19
Effect	(W)	1.70	**1.54**	1.85	1.74
Reward	-	-1.00	**-0.86**	-0.92	-0.98

Table 5.3: Algorithm comparison on MobilenetV2. The ref column describes the properties of the original network and the remaining columns describe the properties of the network when compressed with different algorithms: channel pruning (CP), filter pruning (FP) and Tucker decomposition (TD).

Table 5.3 displays the compression rates achieved on MobileNet-V2, where the highest reward of -0.86 was achieved with the channel pruning action. With this action, the top-1 error was increased by 0.8 and the top-5 error was increased by 0.44. The highest compression rate was achieved for FLOP(s), which was reduced by 19%, and the lowest was achieved for max effect, which was reduced by 9%. The second highest reward of -0.92 was achieved with filter pruning. This reward turned out to be very optimistic as the top-1 error increased by 43.3 on the test set, which is well beyond the acceptable loss region. As for the other two networks, Tucker decomposition yielded a NoOp policy where the reference network was left intact.

5.2 Resource-constrained Optimization

In this experiment, we compare the compression actions according to the resource-constrained protocol. For each type of action, the agent is first warmed up with 50 spidoes of random exploration and then trained for 150 episodes of exploitation. For this experiment, we utilize the full action space $a \in [0, 1.0)$.

Table 5.4 displays the compression rates achieved on ResNet-50. For this network, the best reward of 89.1 was achieved with the channel pruning action. Note that the top-1 training error of 10.9% is substantially higher than the top-1 test error of 3.95%. In fact, the top-1 and top-5 errors decreased by 1.24 and 1.33 on the test set. The highest compression rate was achieved for loading time, which was reduced by 57%, and the lowest was achieved for max effect, which was reduced by 4%. With filter pruning, the agent received a much smaller reward of 34.5 which turned out to be optimistic as the top-1 error was 79.5 on the test set.

Metric	Unit	Ref	CP	FP	TD
Top-1 Error	(%)	5.19	**3.95**	79.5	100
Top-5 Error	(%)	2.23	**0.90**	55.0	100
Model size	(MB)	102	62.1	76.7	**41.2**
Parameters	(M)	25.5	15.5	19.1	**10.2**
FLOP(s)	(G)	6.25	4.78	5.48	**3.65**
FLOP/s	(G/s)	**447**	372	400	292
Throughput	(FPS)	66.4	78.8	72.9	**80.1**
Latency	(ms)	33.8	26.8	**26.3**	27.1
Loading time	(ms)	24.7	10.7	17.6	**9.3**
Init. time	(s)	13	22	22	**12**
Energy	(J)	0.35	0.33	0.31	**0.30**
Effect	(W)	4.32	**4.16**	4.31	**4.16**
Reward	-	-50	**89.1**	34.5	0.00

Table 5.4: Algorithm comparison on ResNet50. The ref column describes the properties of the original network and the remaining columns describe the properties of the network when compressed with different algorithms: channel pruning (CP), filter pruning (FP) and Tucker decomposition (TD).

While filter pruning action achieved the largest reduction of inference latency, the other reductions were not as large as the ones achieved with the other actions. Tucker decomposition achieved the largest compression rates across most performance metrics. However, both top-1 and top-5 error was increased to 100%.

Table 5.5 displays the compression rates achieved on ResNet-18. For this network, the best reward of 0 was achieved by Tucker decomposition. This action also achieved the highest compression rates across most performance metrics. The largest compression rate was achieved for loading time, which was reduced by 80%, and the lowest was compression rate was achieved for energy consumption, which was reduced by 5%. While Tucker decomposition achieved the largest compression rates overall, the top-1 and top-5 errors were increased to 100%. Neither channel pruning or filter pruning managed to compress the network down to 0.8 times of its original size which resulted in flat rewards of -50.

Table 5.6 displays the compression rates achieved on MobileNet-V2. For this network the best reward of 70.5 was achieved with the channel pruning action. The training top-1 error of 29.5 turned out to be slightly pessimistic with respect to the top-1 test error of 21.9. With this action, the largest improvement was achieved on throughput, which was increased by 44%, and the smallest improvement was achieved on energy consumption, which was reduced by 6%. Filter pruning achieved lower compression rates than channel pruning and with 71.5% top-1 error on the training set. This estimate turned out to be optimistic as the top-1 test error turned out to be 100. With Tucker decomposition, the values for many metrics were increased while the top-1 error increased to 100%.

Metric	Unit	Ref	CP	FP	TD
Top-1 Error	(%)	**20.5**	25.5	59.8	100
Top-5 Error	(%)	**4.5**	11.6	34.4	100
Model size	(MB)	48.8	42.6	44.1	**11.4**
Parameters	(M)	11.7	10.6	11.0	**2.83**
FLOP(s)	(G)	1.82	1.62	1.76	**0.93**
FLOP/s	(G/s)	291	**386**	376	236
Throughput	(FPS)	153	238	213	**254**
Latency	(ms)	21.0	14.8	15.0	**12.9**
Loading time	(ms)	11.93	11.23	11.11	**2.50**
Init. time	(s)	**12**	22	22	**12**
Energy	(J)	0.19	**0.17**	0.19	0.18
Effect	(W)	2.01	2.01	2.31	**1.85**
Reward	-	-50	-50	-50	**0.00**

Table 5.5: Algorithm comparison on ResNet18. The ref column describes the properties of the original network and the remaining columns describe the properties of the network when compressed with different algorithms: channel pruning (CP), filter pruning (FP) and Tucker decomposition (TD).

Metric	Unit	Ref	CP	FP	TD
Top-1 Error	(%)	**18.8**	21.9	100	100
Top-5 Error	(%)	4.02	**3.13**	100	97.8
Model size	(MB)	14.2	**9.71**	12.0	30.1
Parameters	(M)	3.50	**2.38**	2.94	7.5
FLOP(s)	(G)	0.72	**0.51**	0.58	2.07
FLOP/s	(G/s)	128	129	140	**204**
Throughput	(FPS)	176	**254**	239	98.4
Latency	(ms)	14.0	**10.7**	12.0	26.3
Loading time	(ms)	3.14	**2.53**	2.81	6.31
Init. time	(s)	**12**	17	17	**12**
Energy	(J)	0.18	**0.17**	**0.17**	0.23
Effect	(W)	1.70	**1.54**	1.85	2.47
Reward	-	-50	**70.5**	28.5	-50

Table 5.6: Algorithm comparison on MobilenetV2. The ref column describes the properties of the original network and the remaining columns describe the properties of the network when compressed with different algorithms: channel pruning (CP), filter pruning (FP) and Tucker decomposition (TD).

5.3 Runtime Optimization

In this experiment, we compare the results obtained by the compression algorithms with those obtained by TensorRT. For brevity, we limited this experiment to channel pruning and Tucker decomposition for the resource constrained optimization. We also focus our evaluations on three performance metrics: throughput, model size and energy consumption.

5.3.1 Channel Pruning and TensorRT

Figure 5.1 shows how the throughput of the networks change with channel pruning and TensorRT. For ResNet-50, the throughput of the pruned network is 20% higher than the reference network. TensorRT managed to increase the throughput of the reference network by almost 4×. The optimal throughput of 357 was achieved when applying TensorRT to the pruned network, which is 45% higher than what TensorRT could achieve on its own. For ResNet-18 and MobileNet-V2, the effect of channel pruning was larger: 55% for ResNet-18 and 44% for MobileNet-V2. However, applying TensorRT to the pruned versions resulted in marginal gains compared to applying it on the original versions.

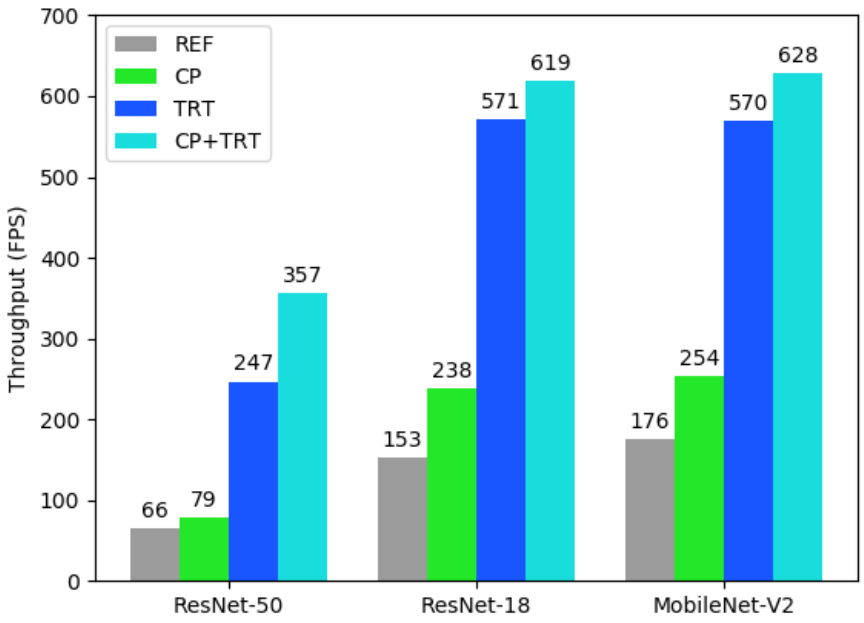

Figure 5.1: Throughput evaluation of pruned networks with and without TensorRT.

Figure 5.2 shows how the size of the networks change with channel pruning and TensorRT. For ResNet-50, the size of the pruned network is 34% smaller than the reference network and the size of the TensorRT version is 24% smaller. The smallest size of 51 MB was achieved when applying TensorRT to the pruned network. For ResNet-18, the comparative order of the techniques is the same but the magnitude of the differences is smaller. For MobileNet-V2, TensorRT achieved a slightly larger size reduction than channel pruning, but the ultimate size reduction was achieved by using both channel pruning and TensorRT.

Figure 5.3 shows how the energy consumption of the networks change when applying channel pruning and TensorRT. For all networks, TensorRT achieved a much larger reduction than channel pruning for this metric. For ResNet-50 and ResNet-18, a slightly lower energy consumption was achieved by applying TensorRT to the pruned networks, compared to using TensorRT only.

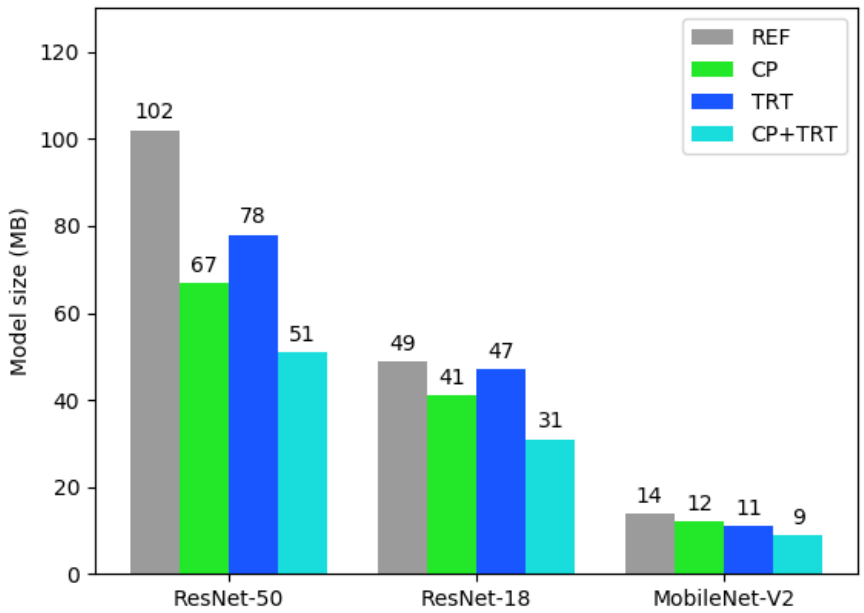

Figure 5.2: Size improvements with and without TensorRT.

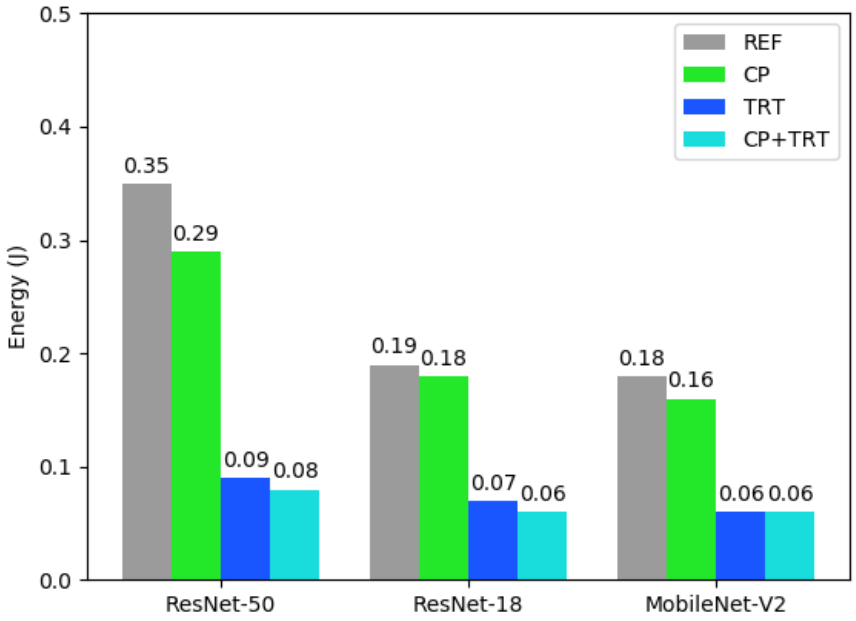

Figure 5.3: Energy improvements with and without TensorRT.

5.3.2 Tucker Decomposition and TensorRT

Figure 5.4 shows how the throughput of the networks change when applying Tucker decomposition, TensorRT and Tucker decomposition plus TensorRT. The patterns for ResNet-50 and ResNet-18 are largely similar to those observed for channel pruning. However, for MobileNet-V2, the throughput decreases significantly when using Tucker decomposition and the optimal throughput is achieved when using just TensorRT.

Figure 5.5 shows how the size of the networks change when applying Tucker decomposition, TensorRT and Tucker decomposition plus TensorRT. The patterns for ResNet-50 and ResNet-18 are somewhat ismilar to those observed for channel pruning, but the magnitude of those reductions is larger. For MobileNet-V2, the size is increased after applying channel pruning and the optimal size is achieved by using just TensorRT.

Figure 5.6 shows how the energy consumption of the networks change when applying Tucker decomposition, TensorRt and Tucker decomposition plus TensorRT. The patterns for ResNet-50 and ResNet-18 are very similar to those observed for channel pruning. However, as for the other metrics, the energy consumption for MobileNet-V2 is increased after applying Tucker decomposition.

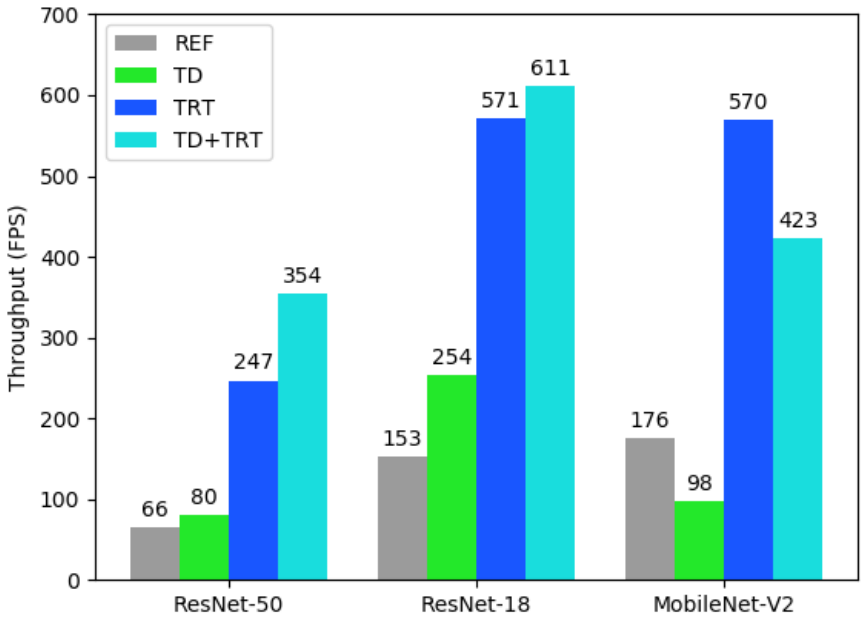

Figure 5.4: Throughput improvements with and without TensorRT.

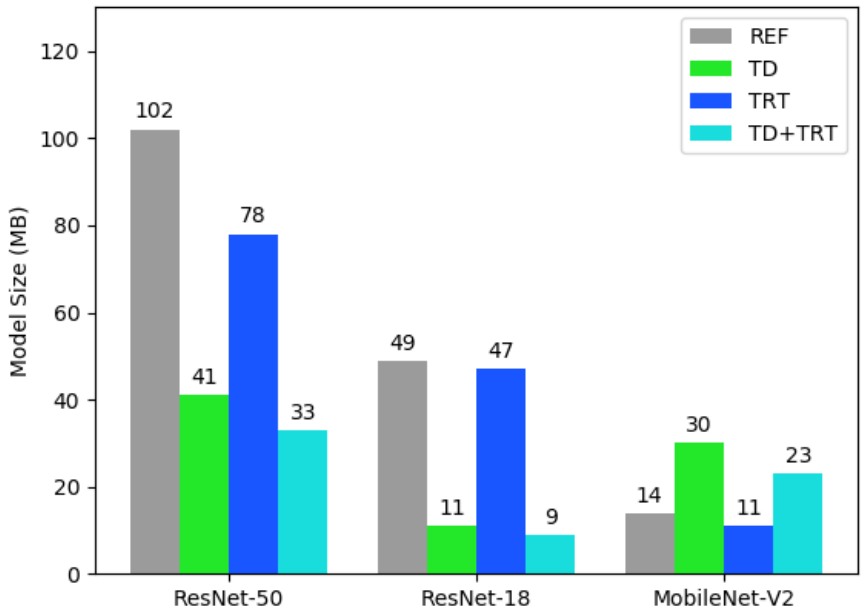

Figure 5.5: Size improvements with and without TensorRT.

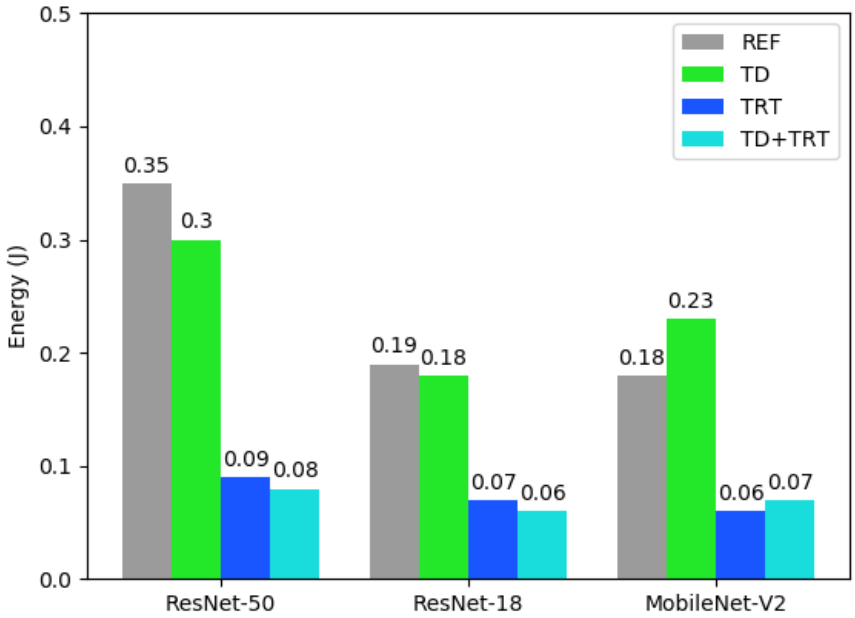

Figure 5.6: Energy improvements with and without TensorRT.

6
Discussion

In the accuracy-guaranteed optimization, channel pruning was the only algorithm that satisfied the accuracy constraint while reducing the resource costs of most metrics. The two metrics it did not manage to improve on were FLOP/s and initialization time. A lower FLOP/s signifies that the computational efficiency is reduced. On the other hand, the throughput was increased, which suggests that data loading became the main bottleneck of the pruned network. The initialization time was increased dramatically for all compressed networks, but not for the ones left intact. The cause of this is not obvious, but one possible explanation is that the used framework may provide optimized serialization for standard (i.e. uncompressed) networks. With filter pruning, we noticed small compression rates for ResNet-50 and MobileNet-V2 and, while the accuracy constraint was satisfied on the training set, it did not generalize well to the test set where the top-1 error increased dramatically. With Tucker decomposition, the agent failed to find an acceptable compression policy and left all networks intact.

In the resource-constrained optimization, no algorithm managed to satisfy the resource constraints on all networks. Channel pruning managed to satisfy the constraints on ResNet-50 and MobileNet-V2, while Tucker decomposition managed to satisfy them on ResNet-50 and ResNet-18. Compared with the other two networks, ResNet-18 consists of fewer convolutional layers and even fewer pairs of such layers. Since pruning is applied on pairs of convolutional layers, those methods become less effective on this architecture. Tucker decomposition did not have the same problem on ResNet-18 as it can operate on single convolutional layers. However, Tucker decomposition did not manage to satisfy the resource constraints on MobileNet-V2, which already contains a large number of separable convolutions.

When we optimized the networks with TensorRT, we noticed a sharp increase in throughput on all reference networks. Furthermore, using TensorRT on the compressed networks yielded a consistently higher throughput than what was obtained by using it on the reference networks. For size reductions, the compression methods showed a greater capacity than TensorRT on ResNet-50 and ResNet-18. Energy consumption was drastically reduced by TensorRT, but we did not manage to reduce it further by compressing the networks. The general conclusion we draw from this experiment is that the suitability of different optimization strategies depend on the resource metrics of interest. That said, the compression methods do not seem to introduce architectural changes that make the runtime optimization more difficult.

6.1 Metrics Correlation

In much of the previous works on network compression, results are evaluated with proxy metrics, such as the reduction of parameters. An assumption is then made that such proxy metrics will have a tight proportional relationship with direct hardware metrics, such as latency and energy consumption. One of the major contributions of this work is the inclusion of direct hardware metrics in both the optimization and evaluation procedures. Since we have collected a set of performance scores for both proxy and direct metrics, it is worthwhile investigate the actual relationship between different pairs of those. Figure 6.1 shows the correlation between pairs of metrics that were collected in this study. It includes the evaluation scores obtained for all algorithms and networks on both accuracy-guaranteed and resource-constrained optimization, without TensorRT. While the correlation is rather high (positive or negative) for most pairs of metrics, it is rarely 1.0. Because of this, it is seldom possible to predict the value of one metric given the value of another. Had we included values obtained from quantization methods, which alter the bitwidth precision of individual weights, we would most likely have seen different correlation scores between size and parameters as well. Notice that energy consumption and initialization time has a very low correlation with most other metrics. Hence, if the problem instance requires a tight constraint on those values, they must be accurately monitored during the optimization process.

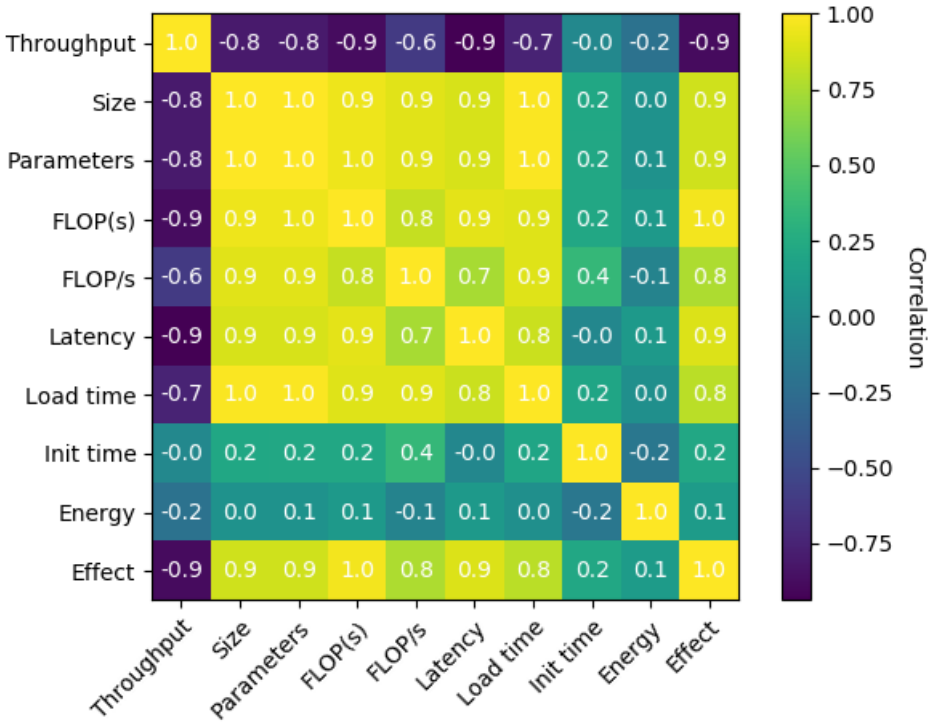

Figure 6.1: Correlations between different performance metrics.

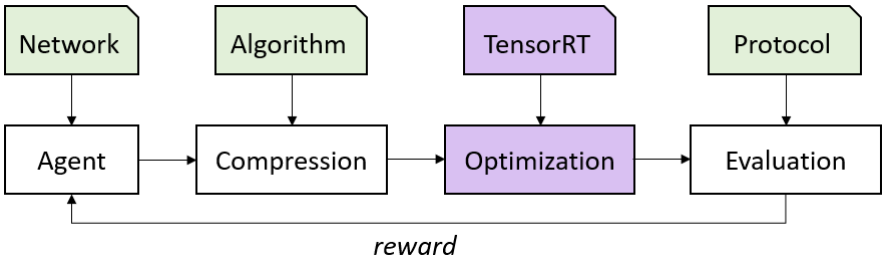

Figure 6.2: The optimization loop with TensorRT added between the compression and evaluation stages.

6.2 Deployment Considerations

To integrate network compression into a continuous software development process, there are some considerations that must be made. The first one is how to utilize TensorRT in the most effective way. In this work, we excluded TensorRT from the optimization loop and only used it after a compression policy had been learned. This was a deliberate decision as it allowed us to analyze the effectiveness of the compression algorithms in isolation. In a real world scenario, this approach comes with the downside that the final compression rates (i.e. with TensorRT) are not known at the evaluation step of the optimization loop. Hence, when optimizing a network with the resource-constrained protocol, the compression policy that is learned may be too aggressive. To remedy this issue, we suggest using TensorRT right before the evaluation step of the optimization loop when training the agent for the resource-constrained optimization. For accuracy-guaranteed optimization, this is much less of an issue since TensorRT will not alter the model accuracy.

The compression policies learned by the reinforcement learning agent can be deployed conveniently by moving the actor and critic networks to the target system. In this work we have trained a separate policy for each network and protocol. This leads to a very bulky deployment, where each network in the target system has to be accompanied by several auxiliary networks. An alternative strategy, which we suggest for future work, is to explore the feasibility of training a single policy for each combination of network and protocol. Since we know from Section 6.1 that many of the metrics are strongly correlated, we believe that this can be done by augmenting the state embedding with information about the network and protocol.

When deploying a compression policy based on channel pruning, one has to devise an efficient strategy for sampling feature maps. Caching feature maps produced by previous inferences will probably yield the fastest compression times. That said, the size of those feature maps may be very large, counteracting the size reductions gained through the compression. An alternative strategy would be to cache images used for previous inferences. This will probably lead to a much lower memory footprint than the first option, but compression times will be slightly longer since the images need to be propagated first. We note that more research is needed for investigating the opportunities for efficient weight reconstruction.

6.3 Ethical Considerations

One ethical implication related to lossy network compression is that the end user may not be aware that the accuracy of certain components change over time. This implication is magnified by the fact that there is no easy way to communicate this information to the user. This can induce a phenomenon known as the *automation expectation mismatch* (AEM), where the user overestimates the autonomous capability of the system. Because of this, it is very important to measure the safety implications of the end system before deploying these techniques. In particular, safety-critical components should only be compressed with a protocol that allows the enforcement of safety-level constraints. In this work, we have proposed the accuracy-guaranteed protocol which possesses this property. While we have focused on top-1 error as the safety-critical metric, the protocol is general enough to support other safety-critical metrics as well. That said, an extensive evaluation of this protocol for other metrics is left for future work.

7
Conclusion

Neural networks are resource hungry devices which make them difficult to deploy to real-time systems with tight resource budgets. Network compression aims to reduce the size and computational complexity of pre-trained neural networks, primarily by reducing the number of parameters. However, the correspondence between the number of parameters and direct hardware metrics, such as latency and throughput, is not always clear. Furthermore, to integrate network compression into safety-critical real-time systems, the optimization algorithms must be able to enforce safety-level constraints.

In this work, we propose splitting the optimization problem into two distinct protocols. In the accuracy-guaranteed protocol, the objective is to minimize resource costs while clamping predictive performance above a specified threshold. In the resource-constrained protocol, the objective is to maximize predictive performance while clamping resource costs below a specified threshold. Each protocol has a trivial solution, which makes it possible to control both safety and resource constraints in systems which consists of many networks.

To solve these protocols, we implemented a reinforcement learning framework that can be specialized into different optimization procedures by changing the compression action. In this work, we considered three different actions: filter pruning, channel pruning and Tucker decomposition. Importantly, no fine-tuning was used in the compression process. We evaluated these methods on ResNet-50, ResNet-18 and MobileNet-V2, all of which had been pre-trained for image classification on the ImageNet dataset. Among the compression actions, we noticed that channel pruning was the most effective as it satisfied almost all constraints posed by the optimization protocols. It only failed on the size constraint on ResNet-18, most probably because of the low number of convolutional pairs in that architecture. Compared to filter pruning, which functions in a similar way, channel pruning has a huge advantage in that it allows for a fast and easy way to reconstruct the feature maps affected by the pruning operation. This allows channel pruning to be applied more aggressively without losing as much accuracy as filter pruning. The results of our experiments also suggest that it can prevent overfitting the compression policy to the training set, which is a phenomenon we observed with filter pruning. Tucker decomposition yielded impressive compression rates on ResNet-50 and ResNet-18, but it showed to be too destructive to the predictive performance of the network. We also showed that the effects gained through pruning and decomposition can, in many cases, be magnified by further optimizing the network with TensorRT.

We have also discussed some of the practical implications of this work. We have shown that, while there exist a correlation between the number of parameters and direct hardware metrics, the correlation is seldom perfect. Hence, if there exists a tight resource budget for the model to be compressed, measured resource costs should be included in the optimization loop. We have also highlighted some important considerations regarding the integration of network compression to software development practices. In particular, for resource-constrained optimization, we think it is important to include runtime inference optimizers (e.g. TensorRT) in the optimization loop to get a better feedback of the resource costs of the final networks.

In the light of the research question, and with previous works in mind, we summarize the main contributions of this work as follows:

- We have proposed an extension of the compression protocols first conceived by He et al. [17], which is more suitable for real-time systems as it supports multiple, direct hardware metrics.

- We have performed an extensive evaluation of common compression approaches, purposefully designed to be applicable for real-time compression. In particular, no fine-tuning or expensive reconstruction method was used. Furthermore, direct hardware metrics were used to evaluate the results.

- We have shown that improvements made by network compression can often be magnified with TensorRT, but the effect of this magnification can vary between network architectures and the metrics of interest.

- We have shown that direct hardware metrics are seldom perfectly correlated, and sometimes very weakly correlated, with the number of parameters of a network.

- We have proposed the inclusion of TensorRT in the optimization loop for a more robust compression process.

7.1 Limitations

One of the main limitations of this work is the slightly degenerate nature of the task that was used to measure the performance of the compression algorithms. First, the predictive range is much larger than the set of classes that exist in the test set. Secondly, the number of samples used, both for optimization and evaluation, is relatively small. Because of this, we want to be careful not to make too wide generalizations of our results. In particular, we do not guarantee that the detailed performance statistics are generalizable to other tasks with different datasets. That said, we believe that the simulated test environment is realistic enough that the high-level conclusions can be generalized to similar tasks within the domain of safety-critical real-time systems.

Another limitation is the small number of iterations used to train the reinforcement learning agent. In this work, we used 50 exploration episodes and 150 exploitation episodes. We believe that slightly better compression rates could have been obtained with several hundred more episodes. That said, we believe that the main conclusions, including the ranking of the algorithms across the combinations of networks and protocols, would have been very similar.

7.2 Future Work

With the previous discussion of limitations in mind, an obvious step forward would be to evaluate the compression methods in the context of a real safety-critical real-time system. For example, it would be valuable, from both an academic and industrial perspective, to measure their performance on object detection systems. Not only are these types of systems ubiquitous in the realm of safety-critical real-time systems, but they are seldom used as a test environment with which to measure the performance of compression algorithms. Similarly, it could be worthwhile to rerun the experiments on more episodes to see whether this has any significant impact.

From an academical point of view, more research is needed to investigate the opportunities for efficient weight reconstruction. In this work, a lest-squares approach was used to reconstruct the feature maps affected by channel pruning. The approach is simple, fast and, as was demonstrated in the results chapter, reasonably effective. One downside of this approach is that it requires feature maps to be sampled in advance and it is difficult to determine the right volume of this sampling. An alternative approach would be to train an auxiliary network to generate an appropriate weight tensor given a compression configuration. This method has been proposed before by [26], but it has not been evaluated in the context of our implementation and the constraints posed by safety-critical real-time systems.

Bibliography

[1] Anubhav Ashok, Nicholas Rhinehart, Fares Beainy, and Kris M Kitani. "N2n learning: Network to network compression via policy gradient reinforcement learning". In: *arXiv preprint arXiv:1709.06030* (2017).

[2] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. "Compressing neural networks with the hashing trick". In: *International conference on machine learning*. 2015, pp. 2285–2294.

[3] François Chollet. "Xception: Deep learning with depthwise separable convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1251–1258.

[4] NVIDIA Corporation. *https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit*. Accessed on 1 June 2020. 2020.

[5] NVIDIA Corporation. *https://docs.nvidia.com/cuda/profiler-users-guide/index.html*. Accessed on 1 June 2020. 2020.

[6] NVIDIA Corporation. *https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/index.htmlpage/Tegra%20Linux%20Driver%20Package%20Development%20Guide/AppendixTegraStats.html*. Accessed on 1 June 2020. 2019.

[7] Li Deng. "The mnist database of handwritten digit images for machine learning research [best of the web]". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[8] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando De Freitas. "Predicting parameters in deep learning". In: *Advances in neural information processing systems*. 2013, pp. 2148–2156.

[9] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. "Exploiting linear structure within convolutional networks for efficient evaluation". In: *Advances in neural information processing systems*. 2014, pp. 1269–1277.

[10] Samuel Dodge and Lina Karam. "A study and comparison of human and deep learning recognition performance under visual distortions". In: *2017 26th*

international conference on computer communication and networks (ICCCN). IEEE. 2017, pp. 1–7.

[11] Abhimanyu Dubey, Moitreya Chatterjee, and Narendra Ahuja. "Coreset-based neural network compression". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 454–470.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[13] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. "EIE: efficient inference engine on compressed deep neural network". In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 243–254.

[14] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[15] Song Han, Jeff Pool, John Tran, and William Dally. "Learning both weights and connections for efficient neural network". In: *Advances in neural information processing systems*. 2015, pp. 1135–1143.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: `1512.03385 [cs.CV]`.

[17] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. "Amc: Automl for model compression and acceleration on mobile devices". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 784–800.

[18] Yihui He, Xiangyu Zhang, and Jian Sun. "Channel pruning for accelerating very deep neural networks". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 1389–1397.

[19] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network". In: *arXiv preprint arXiv:1503.02531* (2015).

[20] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized neural networks". In: *Advances in neural information processing systems*. 2016, pp. 4107–4115.

[21] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. "Compression of deep convolutional neural networks for fast and low power mobile applications". In: *arXiv preprint arXiv:1511.06530* (2015).

[22] Raghuraman Krishnamoorthi. “Quantizing deep convolutional networks for efficient inference: A whitepaper”. In: *arXiv preprint arXiv:1806.08342* (2018).

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[24] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[25] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. “Continuous control with deep reinforcement learning”. In: *arXiv preprint arXiv:1509.02971* (2015).

[26] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Kwang-Ting Cheng, and Jian Sun. “Metapruning: Meta learning for automatic neural network channel pruning”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 3296–3305.

[27] Gabriel L Oliveira, Wolfram Burgard, and Thomas Brox. “Efficient deep models for monocular road segmentation”. In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2016, pp. 4885–4891.

[28] Siddharth Roheda and Hamid Krim. *Conquering the CNN Over-Parameterization Dilemma: A Volterra Filtering Approach for Action Recognition*. 2019. arXiv: `1910.09616 [cs.CV]`.

[29] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

[30] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[31] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. “Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 764–775.

[32] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[33] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[34] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. "Mnasnet: Platform-aware neural architecture search for mobile". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2820–2828.

[35] Denis Tomè, Federico Monti, Luca Baroffio, Luca Bondi, Marco Tagliasacchi, and Stefano Tubaro. "Deep convolutional neural networks for pedestrian detection". In: *Signal processing: image communication* 47 (2016), pp. 482–489.

[36] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Själander. "BISMO: A scalable bit-serial matrix multiplication overlay for reconfigurable computing". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2018, pp. 307–3077.

[37] Han Vanholder. *Efficient Inference with TensorRT*. 2016.

[38] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. "Haq: Hardware-aware automated quantization with mixed precision". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2019, pp. 8612–8620.